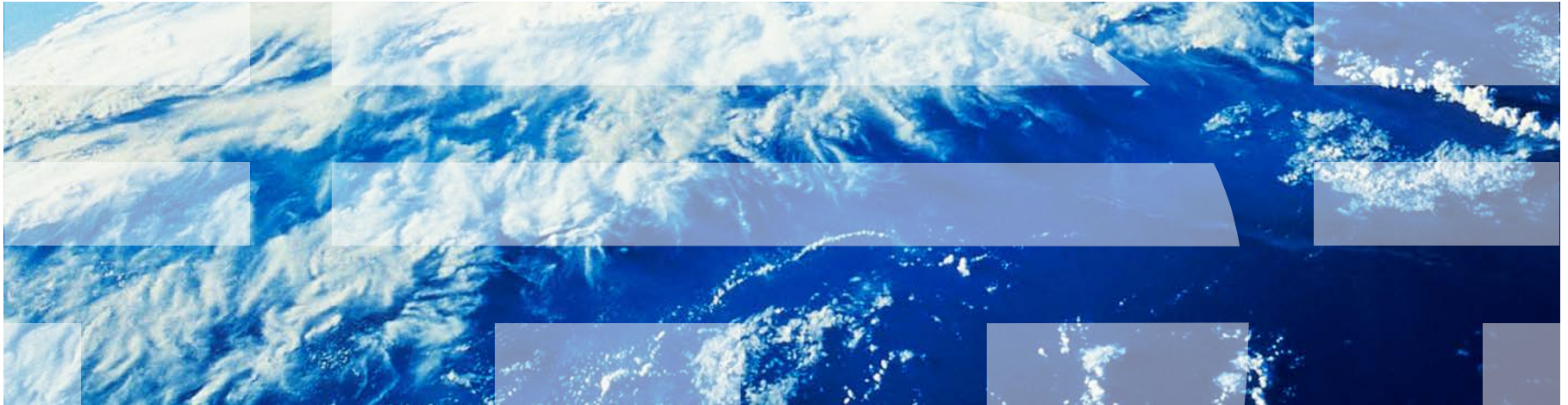


---

# Computer Systems for Data Science

## Topic 4

**Memory/Storage Hierarchy**  
**Database and Key-value Store Architecture**



---

## What we'll cover

- **Memory and storage hierarchy and trade offs**
  - DRAM
  - Flash
  - Disk
  - NVM
- **Indexing**
- **Bloom filters**
- **Key-value stores**
  - RocksDB
- **Database on top of key-value store**
  - MyRocks

---

# Tasks of the database

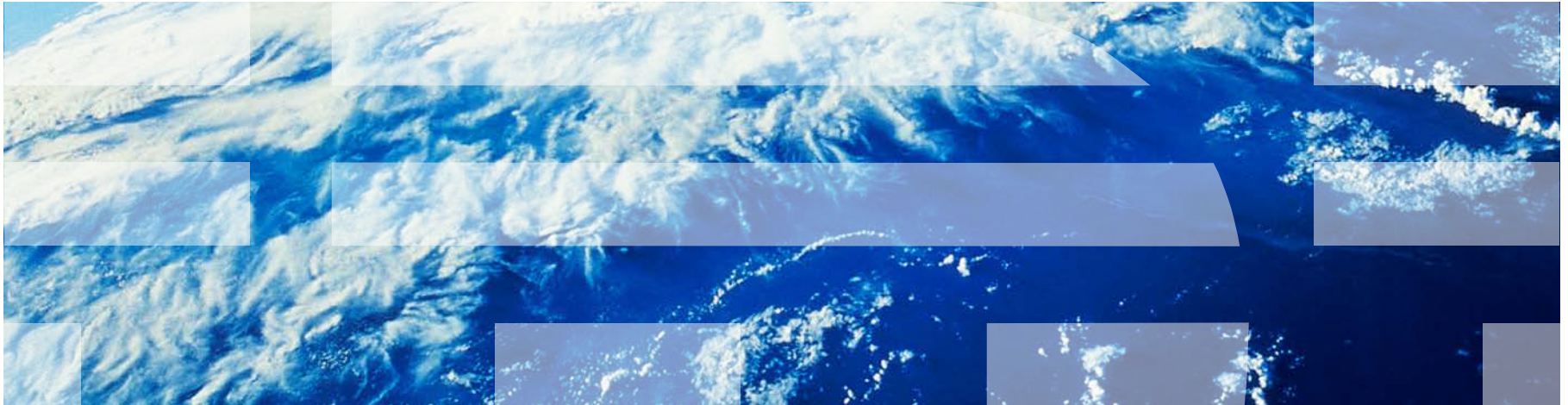
- Low level tasks:
  - Store the tables
    - Example: How do I physically store the tables on disk?
  - Read/insert/delete/update data
    - Example: What is the entry of cuid 1212?
- Higher level semantics:
  - Logging
  - Concurrency control
    - Example: what's the best schedule to execute a set of transactions?
  - Query optimization
    - Example: should I first do the projection, or the aggregation?

---

## We'll focus on the systems aspects of some of these questions

- Low level tasks:
  - **Store the tables**
    - Example: How do I physically store the tables on disk?
  - **Read/insert/delete/update data**
    - Example: What is the entry of cuid 1212?
- Higher level semantics:
  - **Logging**
  - Concurrency control
    - Example: what's the best schedule to execute a set of transactions?
  - Query optimization
    - Example: should I first do the projection, or the aggregation?

# Memory and Storage Hierarchy

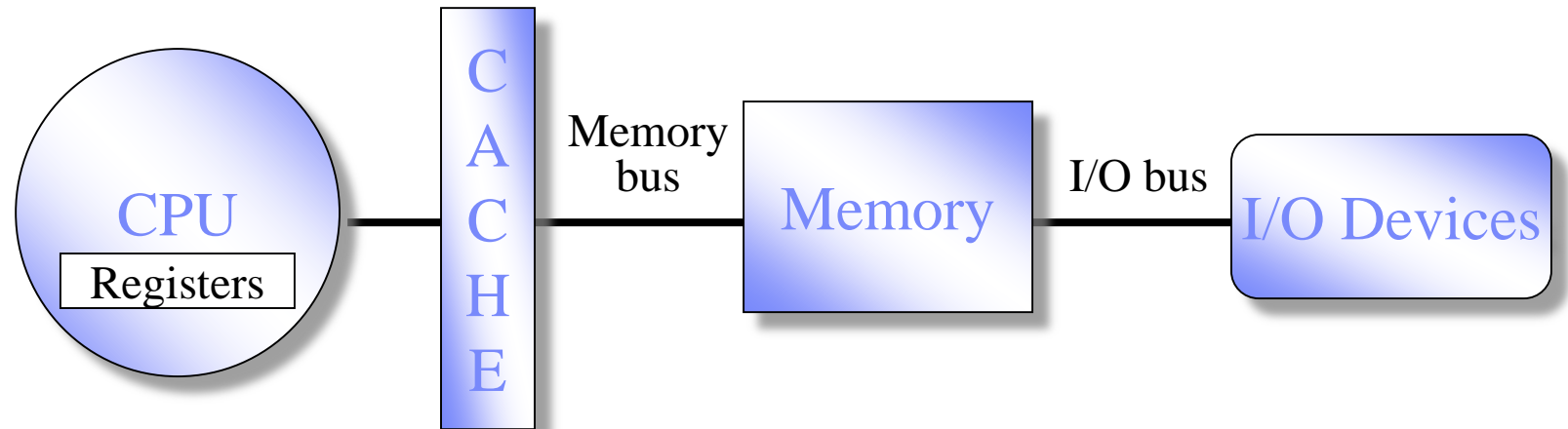


---

## The memory hierarchy

- Memory and storage present several trade-offs:
  - Cost
  - Latency and throughput
  - Durability/volatility (is information lost when power goes down?)
  - Access granularity
- Can't have it all!

## The memory hierarchy



Register  
Reference

L1/L2/L3 Cache  
Reference

Memory  
Reference

Disk  
Reference

Size: 300 B  
Speed: 0.2 ns

1 / 4 / 20MB  
1 / 5 ns

8-128 GB  
25-100 ns

1-3 TB  
5-10 ms  
(smaller, faster  
for Flash SSD)

## Trade-offs (rough numbers)

	Memory (RAM)	Flash	Magnetic Disk
Cost/GB	\$3	\$0.1	\$0.04
Latency, random read	25-100ns	50-100us (1000X RAM)	5-10ms (100X Flash)
Bandwidth, sequential reads	10GB/s	250MB/s	150MB/s
Durable?	No	Yes	Yes
Effective access granularity	Byte reads and writes	4KB read MBs write	MBs read and write

Sources:

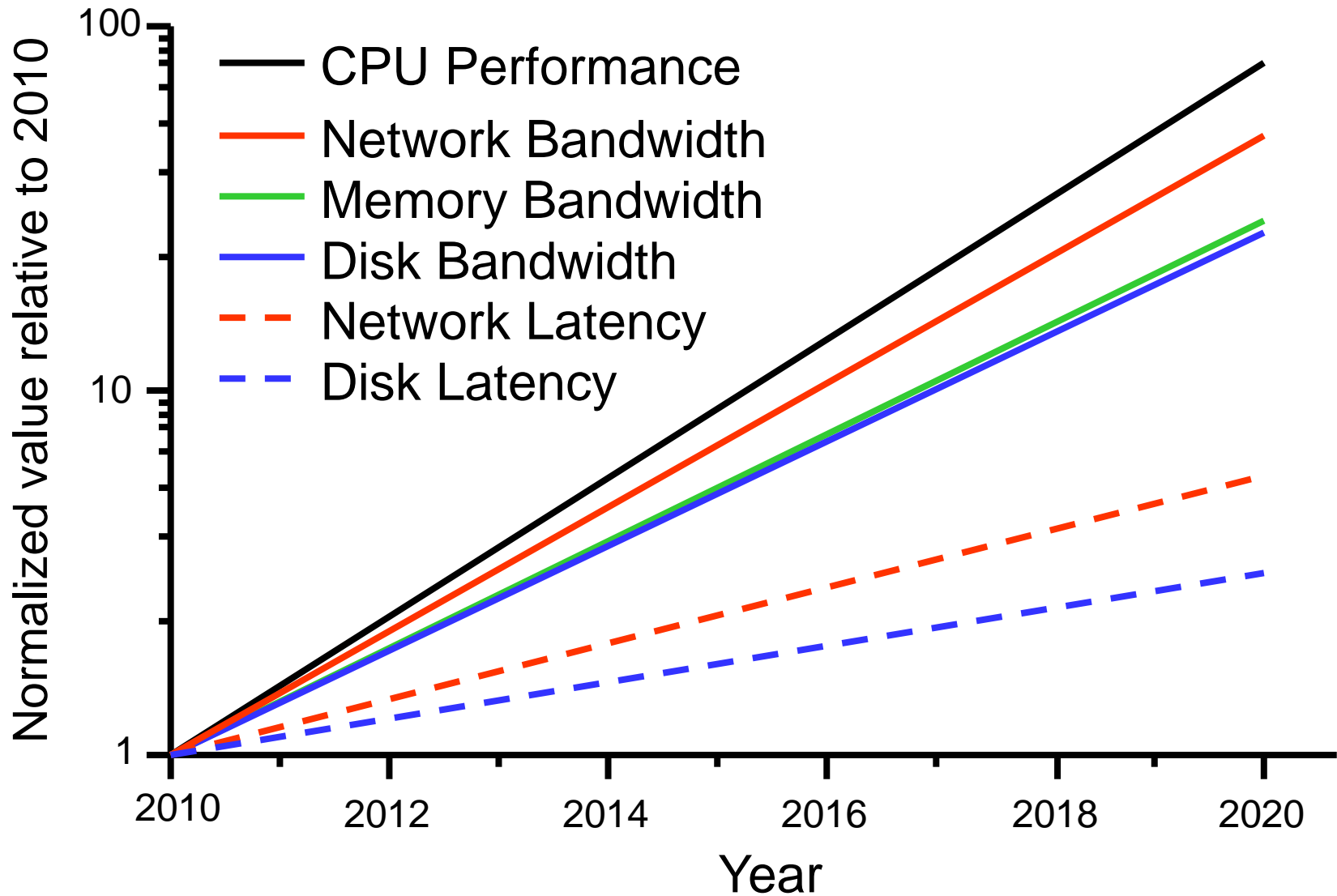
<https://jcmr.net/memoryprice.htm>,

<https://www.amazon.com/Intel-660p-1-0TB-80mm-978350/dp/B07GCL6BR4>

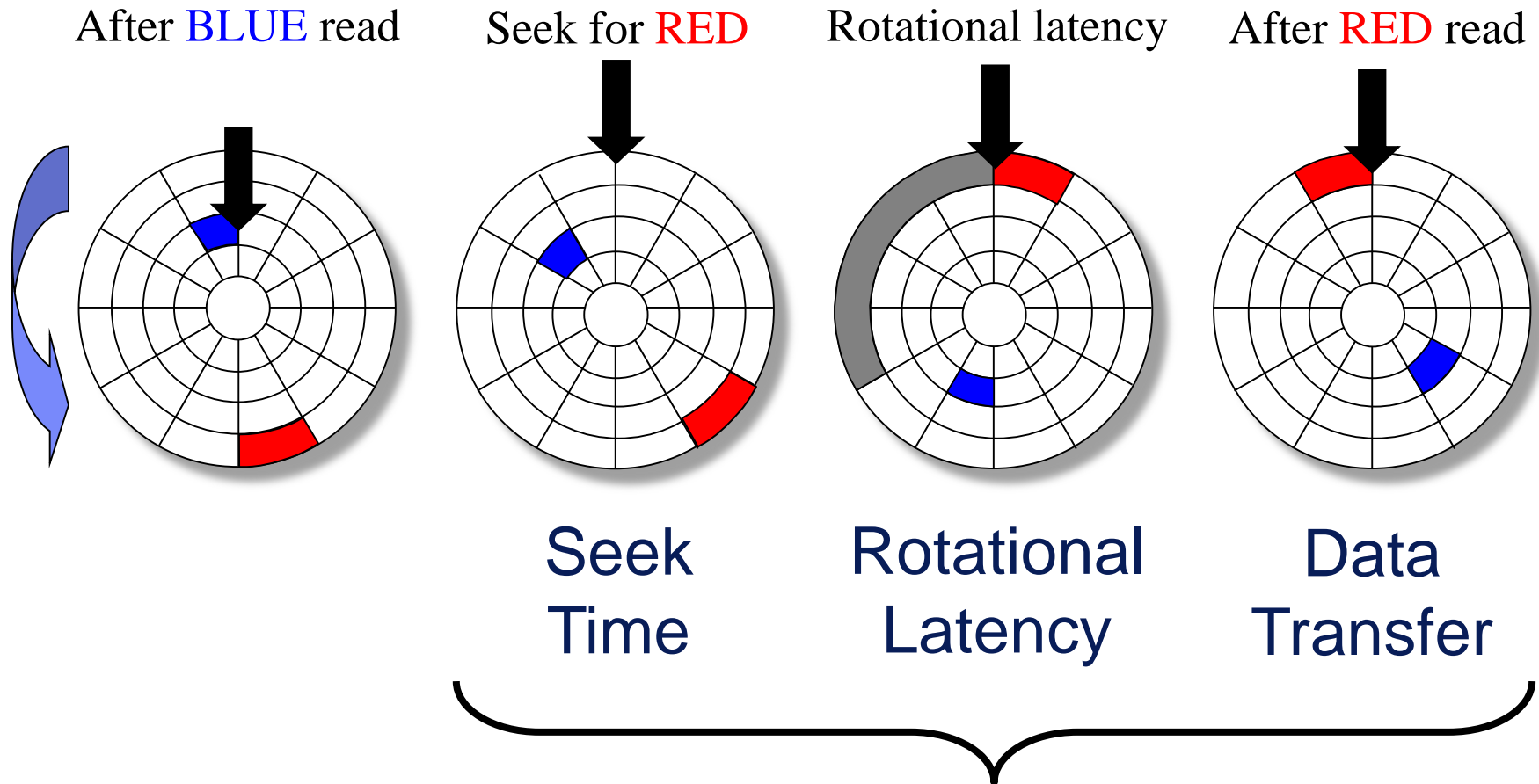
[https://www.amazon.com/Seagate-Portable-External-Hard-Drive/dp/B07CRG7BBH/ref=sr\\_1\\_3?keywords=hard+disk+1tb&qid=1579102708&sr=8-3](https://www.amazon.com/Seagate-Portable-External-Hard-Drive/dp/B07CRG7BBH/ref=sr_1_3?keywords=hard+disk+1tb&qid=1579102708&sr=8-3)



## Technology Trends



## HDD / Magnetic Disk



Access Time = seek time + rotation time  
+ data transfer time  
(for RED)

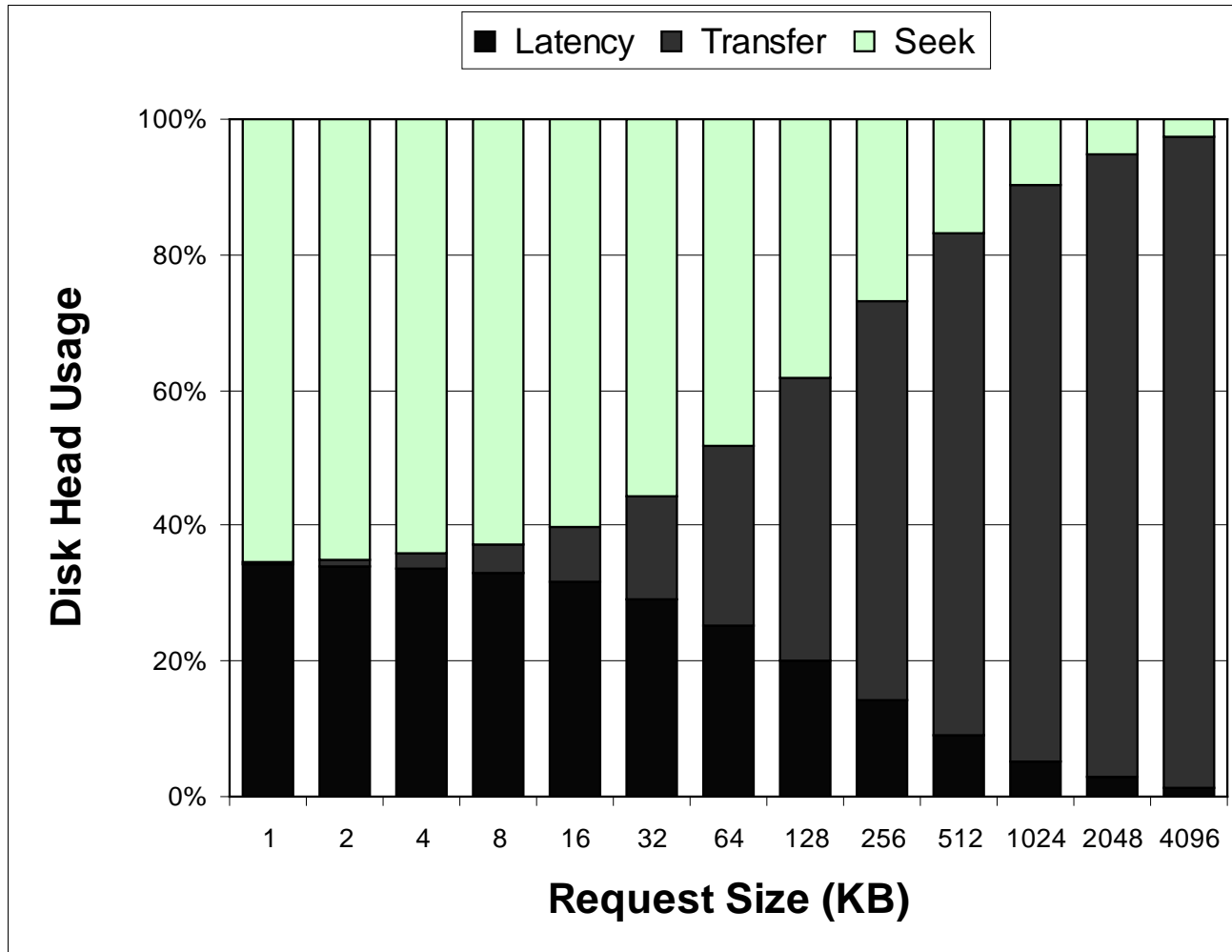
---

## Modern HDD performance

- Seek times: 1 – 6 ms, depending on distance
  - Average 2 – 4 ms
  - Not necessarily improving
    - It's a mechanical part!
- Rotation speeds: 4,200 – 15,000 RPM
  - Average latency of 2-12 ms
  - Typically slowing down
    - But also mechanical
- Data rates: 60-125 MB/s\*, depending on zone
  - avg sector transfer time of 25 us
  - improving at 0-20+% per year

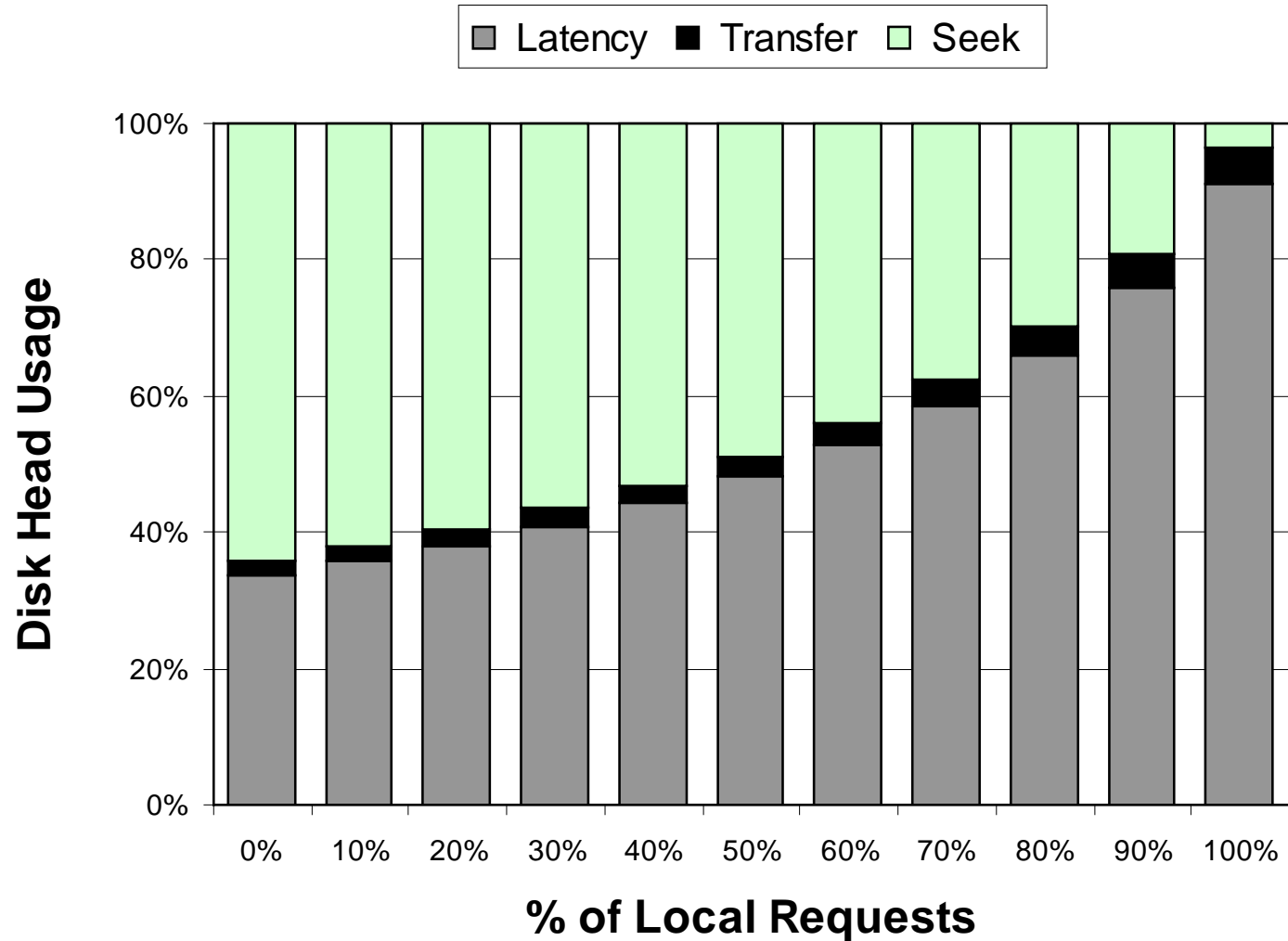


## HDD utilization (random requests)





## Locality is important (4KB requests)



---

# Flash

- Solid-state storage technology
  - Non-volatile memory chips, disk interface, **no mechanical moving parts (yay!)**
  - Fast access (no seeks) but more costly than mechanical disks
  - Lower power (~none if not accessing), noise free (laptops!)
- Performance differences
  - No seeks means high rates of small random access
- Reliability differences
  - Device fatigue (eventually stops being programmable)
  - Device retention (information leaks away)



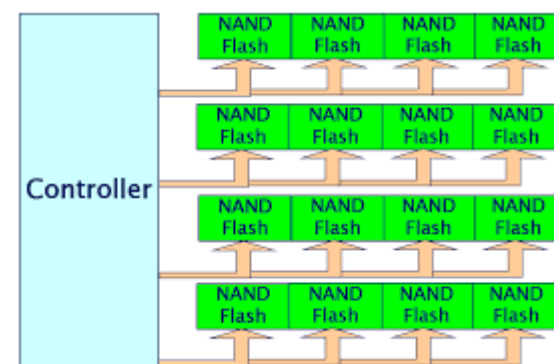
## Flash performance is very different than magnetic disk

---

- Small random reads (no moving parts)
  - much, much lower average latency than mechanical disks
    - e.g., 10s of microseconds
  - orders of magnitude higher throughput than mechanical disks
    - e.g., 10s to 100s of 1000s of operations per second
- Write performance is more complicated
  - Small writes usually 100s of microseconds
    - Need to erase and reprogram flash cells → much longer time than reads
  - Write throughput more lower than reads for small objects
  - Need to write in large contiguous chunks (tens-hundreds of MBs) to get good performance

## Flash prefers large sequential writes

- Large sequential writes perform better on flash. Why?
- **Reason 1:** Erase granularity  $\neq$  write granularity
  - In order to write new data to flash, data needs to be erased first
  - Erasures are done in a block granularity:
    - Block is 4-16KB
  - Writing even 1 bit requires rewriting 4-16KB!
- **Reason 2:** Flash uses multiple channels to get high throughput
  - Writes are *multiplexed* across multiple chips in parallel for high throughput
  - A single logical written block gets chopped up into many physical blocks



For optimal performance: writes typically need to be > 8MB!

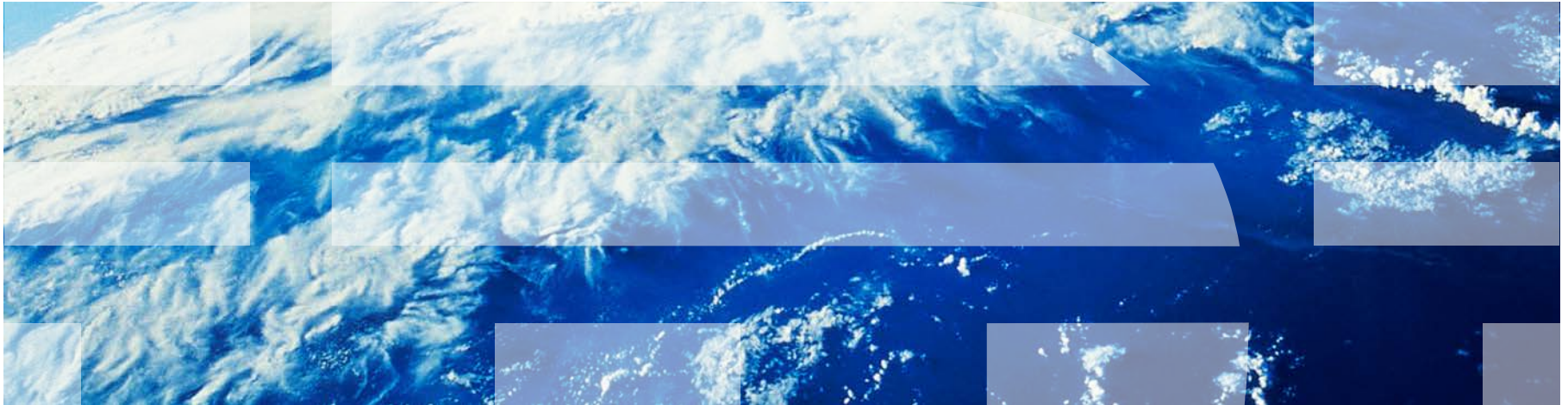


---

## Wear leveling: spread writing evenly in SSD

- Each flash cell can only survive a given number of writes
  - 500 (USB stick), 3,000-100,000 (SSD drives)
- Each block wears independently, so a heavily written block can wear out long before a mostly-read block
  - *Wear leveling* is remapping of addresses to better balance the number of write cycles seen by each block
  - This is done transparently by the storage device, but can lead to unexpected performance drops

# Indexing





# Indexing

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index** consists of records (called **index entries**) of the form

search-key	pointer
------------	---------

- The index is typically much smaller than the original data
  - E.g., 100X smaller
- Where would you store the index?
  - Usually in memory
  - → Index has to be small, since memory capacity is limited

---

## Index Evaluation Metrics

- Access types supported efficiently:
  - Records with a specified value in the attribute
  - Records with an attribute value falling in a specified range of values
- Access time
- Insertion time
- Deletion time
- Space overhead

---

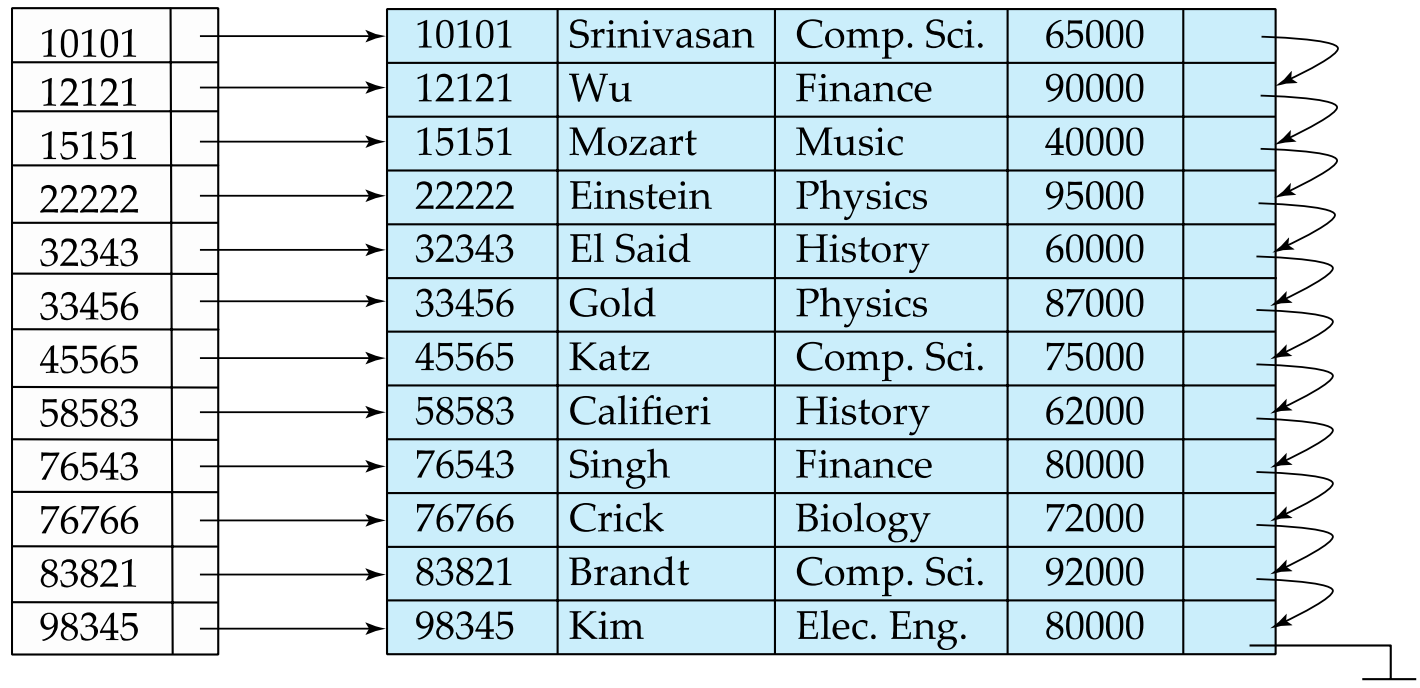
## Ordered Indices

- In an **ordered index**, index entries are stored sorted on the search key value.
- **Primary index**: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file.

## Dense Index Files

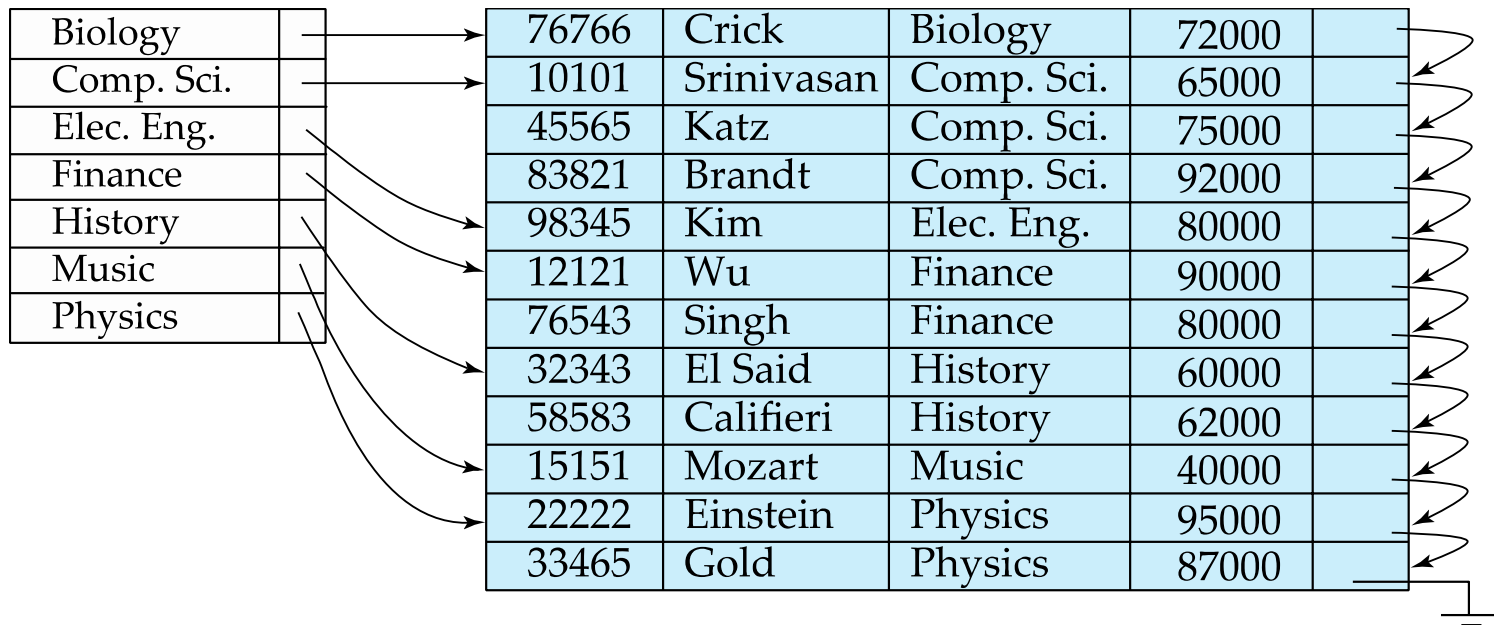
- **Dense index** — Index record appears for every search-key value in the database.
- E.g. index on *ID* attribute of *instructor* relation

10101	→	10101	Srinivasan	Comp. Sci.	65000	→
12121	→	12121	Wu	Finance	90000	→
15151	→	15151	Mozart	Music	40000	→
22222	→	22222	Einstein	Physics	95000	→
32343	→	32343	El Said	History	60000	→
33456	→	33456	Gold	Physics	87000	→
45565	→	45565	Katz	Comp. Sci.	75000	→
58583	→	58583	Califieri	History	62000	→
76543	→	76543	Singh	Finance	80000	→
76766	→	76766	Crick	Biology	72000	→
83821	→	83821	Brandt	Comp. Sci.	92000	→
98345	→	98345	Kim	Elec. Eng.	80000	→



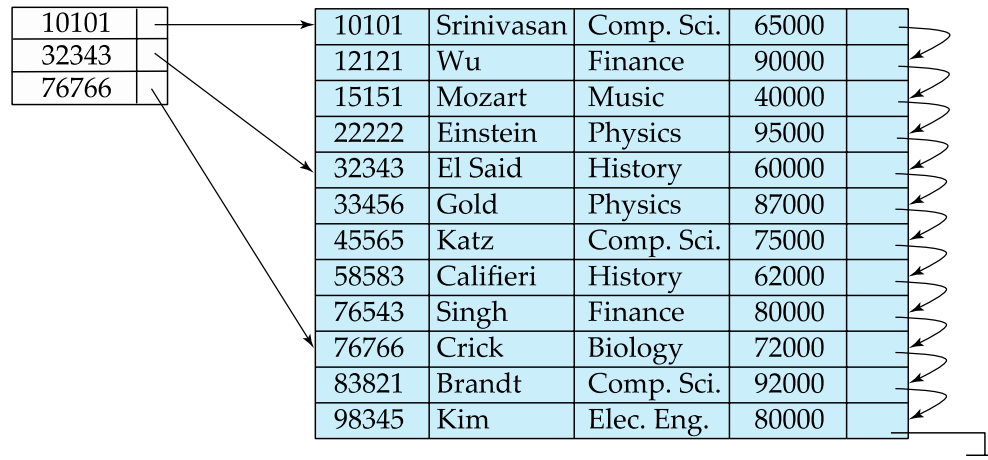
## Dense Index Files (Cont.)

- Dense index on *dept\_name*, with *instructor* file sorted on *dept\_name*



## Sparse Index Files

- **Sparse Index:** contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value  $K$  we:
  - Find index record with largest search-key value  $< K$
  - Search file sequentially starting at the record to which the index record points

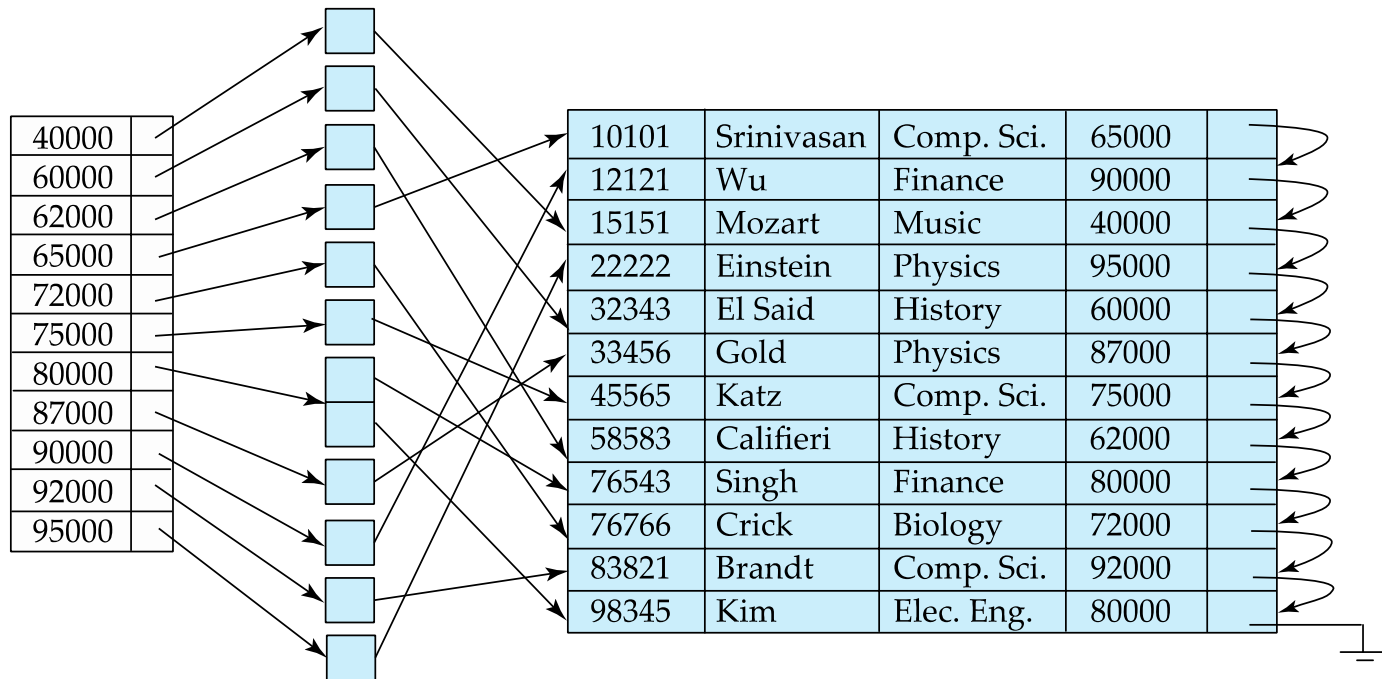


- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions
  - Generally slower than dense index for locating records



## Secondary Index Example

- Secondary index on salary field of instructor



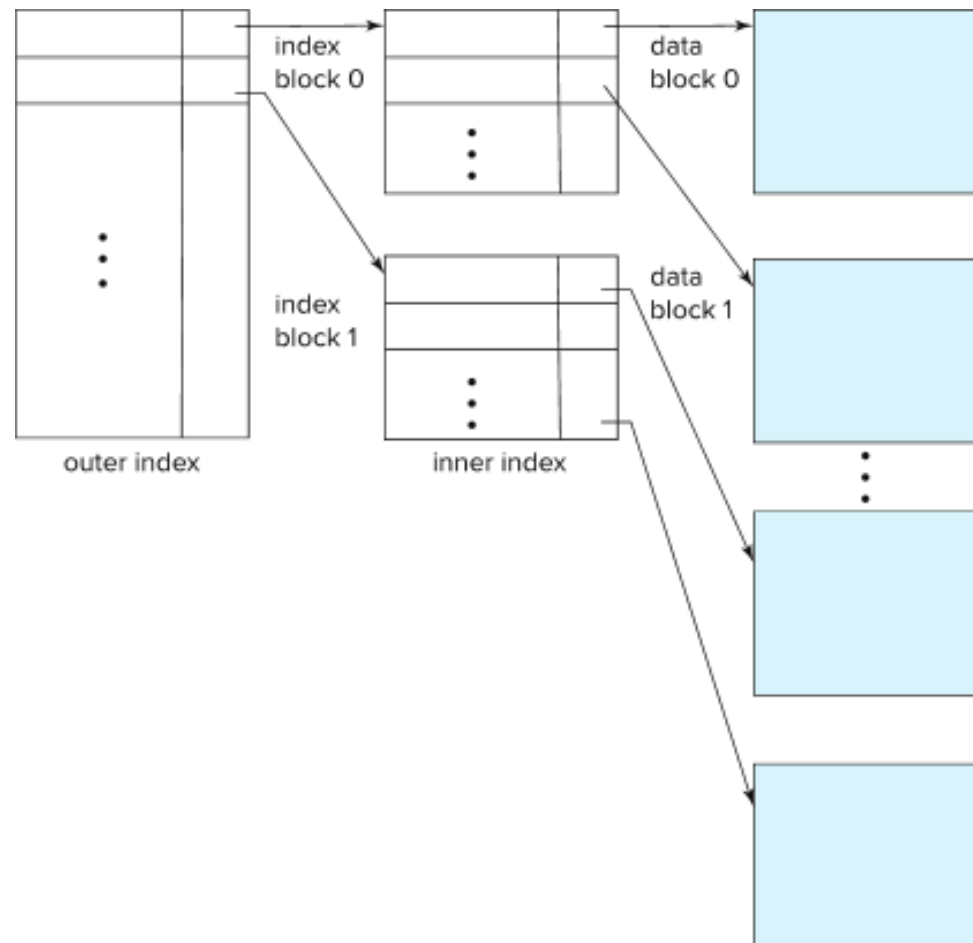
- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense
  - Why?

---

## Multilevel Index

- If index does not fit in memory, access becomes expensive.
- Solution: treat index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of the basic index
  - inner index – the basic index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.
- File systems often use a multilevel index (e.g., nested directories)

## Multilevel Index (Cont.)



---

## Index Update: Insertion

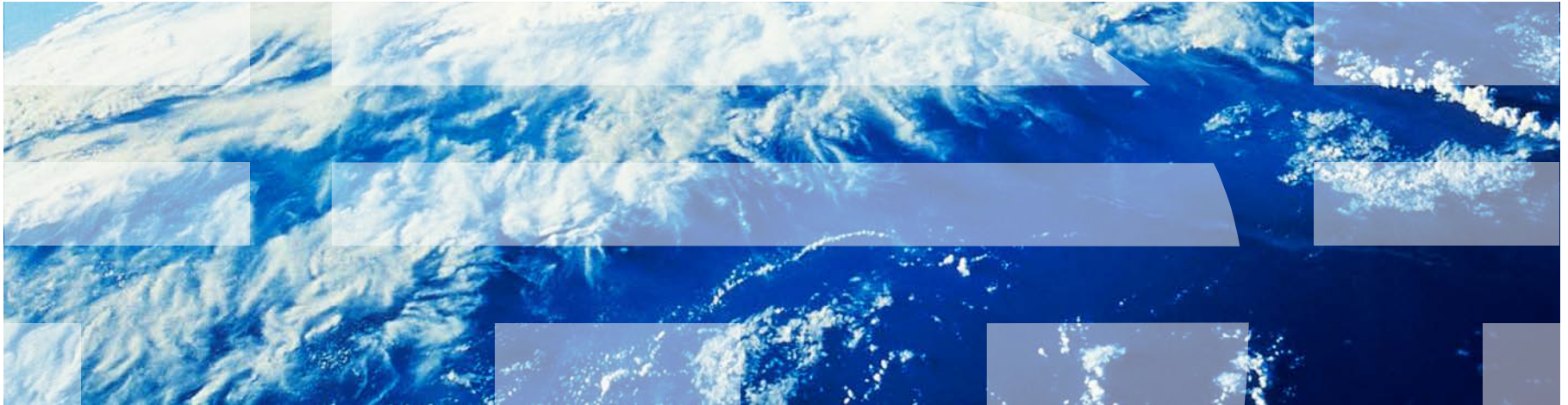
- **Single-level index insertion:**

- Perform a lookup using the search-key value of the record to be inserted.
- **Dense indices** – if the search-key value does not appear in the index, insert it
  - Indices are maintained as sequential files
  - Need to create space for new entry, overflow blocks may be required
- **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
  - If a new block is created, the first search-key value appearing in the new block is inserted into the index.

- **Multilevel insertion and deletion:** algorithms are simple extensions of the single-level algorithms

---

# Bloom Filter



---

## Motivation: Index doesn't fit in memory

- Example of a typical database:
  - 64GB memory
  - 2TB flash disk
  - Average key+value (e.g., entry) size: 50B → database needs to store 40 billion keys!
- A pointer needs to locate a key among 40 billion keys
  - Minimum pointer size: 36 bits ~ 4.5 bytes (let's round it to 5 bytes)
    - $2^{36} \approx 68$  billion
  - Index size = number of keys \* byte rounded pointer size ~ 200GB
  - Does not fit in memory
- Therefore, we must use a multi-level index
  - Outer index in memory
  - Inner index on disk
- Minimum time reading a single object: 1 memory access + 2 flash accesses ~ 200us
- What if object doesn't exist in the database?
  - Still 200us!
- Can we do better?

---

## Bloom filters [Bloom 1970]:

Approximate way to determine if object exists



---

## Bloom filter parameters

- An array of  $m$  bits (can only be '0' or '1')
  - Array initialized to 0
- $k$  independent **hash functions**  $h_1, h_2, \dots, h_k$  that return a number between  $1, \dots, m$



---

## What is a hash function?

- What is a hash function?
  - $h(x) = y$ , where  $y$  is a uniformly random number
  - In our case,  $y$  is a random number between  $1, \dots, m$
- If  $h(x_1) = h(x_2)$ , there is some probability that  $x_1 = x_2$
- If  $h(x_1) \neq h(x_2)$ , we are sure that  $x_1 \neq x_2$

---

## Algorithms: check membership and add membership

- To check if  $x$  is a member of the bloom filter, check whether  $h_1(x), \dots, h_k(x)$  are all set to 1
  - If not,  $x$  is definitely not a member
  - If yes,  $x$  might be a member
    - There can be false positives!
- To add  $x$ , set the positions of  $h_1(x), \dots, h_k(x)$  all to 1
  - Some positions might already have been set as 1

---

## How does a bloom filter work? Example

$m = 10$

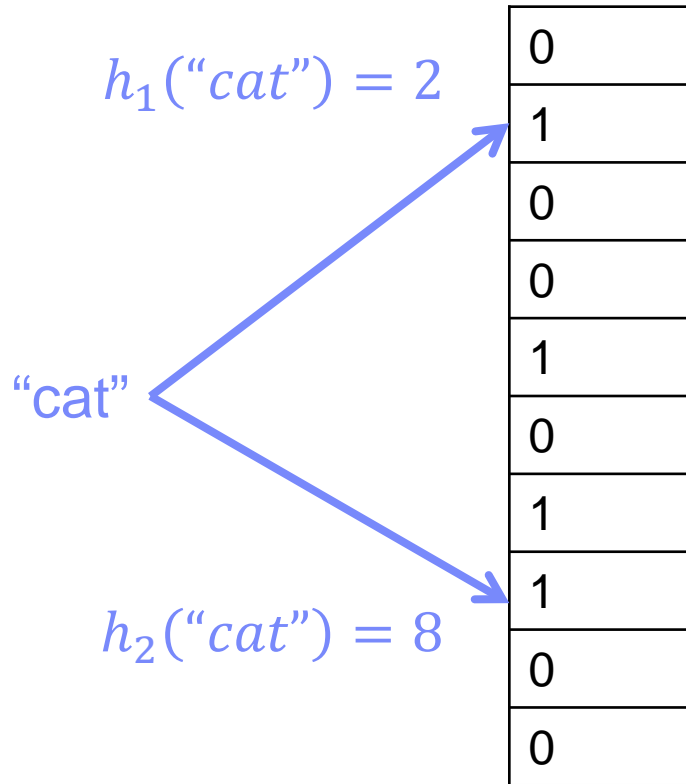
$k = 2$

0
1
0
0
1
0
1
1
0
0

Is cat in DB?

$m = 10$

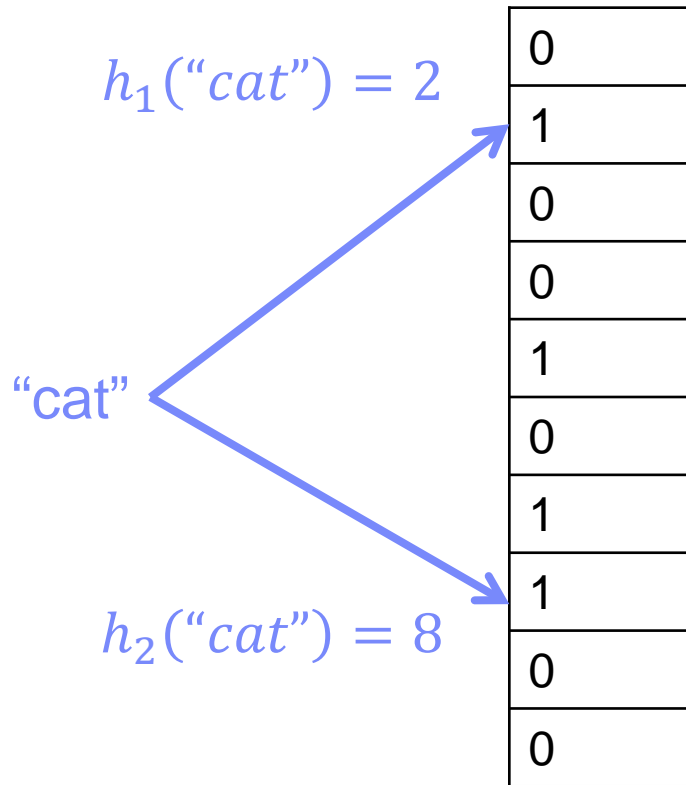
$k = 2$



Is cat in DB? Maybe!

$m = 10$

$k = 2$

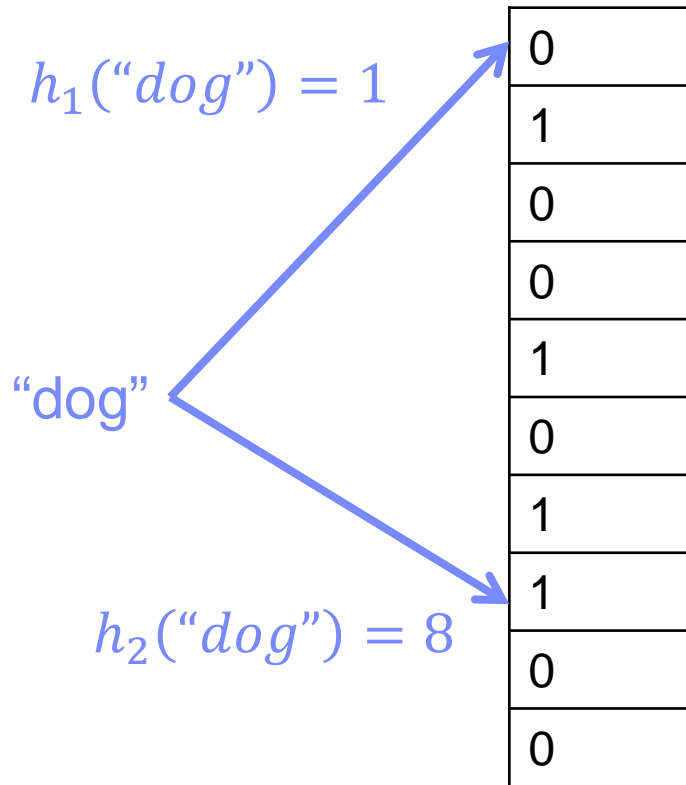


Cat might exists in DB!

Is dog in DB?

$m = 10$

$k = 2$



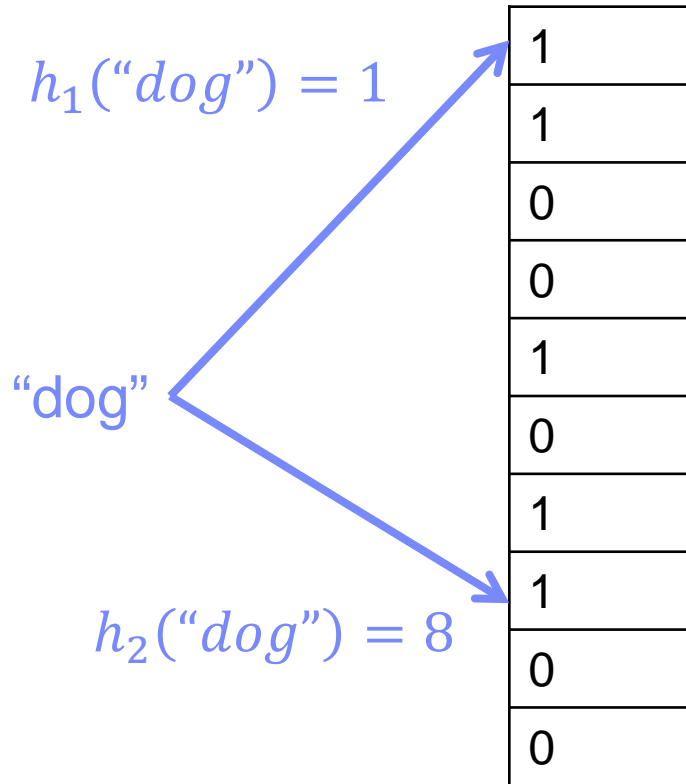
Dog definitely does not exist in DB

→ We don't need to read from disk

## Add dog to DB

$m = 10$

$k = 2$



---

## Goal: low false positives

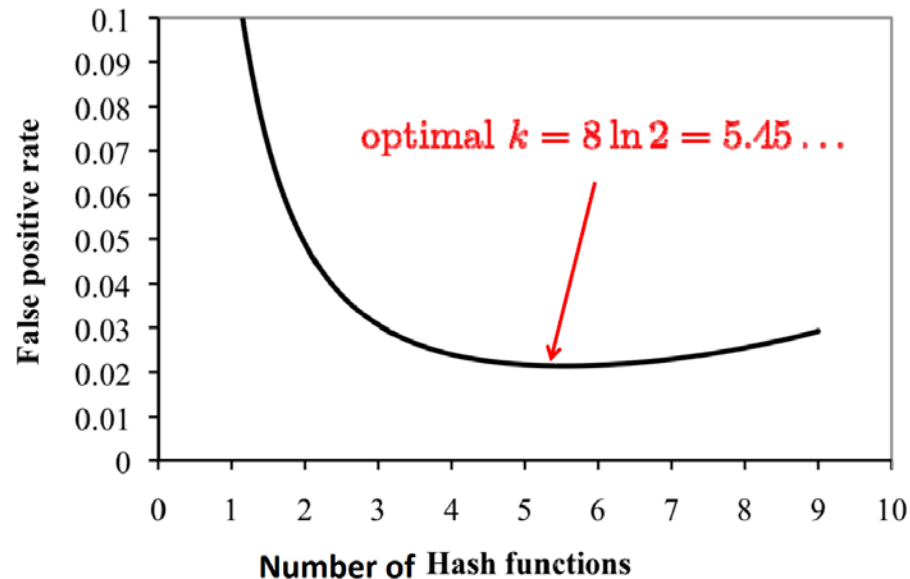
- Define  $n$  as the total number of unique objects that might ever be inserted into the database
  - For example, for a bank database that uses account ID as keys, this is the total number of accounts the bank will ever have
- Let's assume  $kn < m$



## False positive probability (credit: Simon S. Lam)

- The size of  $k$  is a trade-off:
  - A higher  $k$  increase the number of hash functions that might map to 0
  - But also "depletes" the available 0 slots in the bloom filter
- Optimal  $k$ :  $k = \frac{m}{n} \ln 2$

Number of bits per member  $\frac{m}{n} = 8$



# Bloom filter calculator (<https://hur.st/bloomfilter/>)

## ☐ Bloom Filter Calculator ☐

Bloom filters are space-efficient probabilistic data structures used to test whether an element is a member of a set.

They're surprisingly simple: take an array of  $m$  bits, and for up to  $n$  different elements, either test or set  $k$  bits using positions chosen using hash functions. If all bits are set, the element *probably* already exists, with a false positive rate of  $p$ ; if any of the bits are not set, the element *certainly* does not exist.

Bloom filters find a wide range of uses, including tracking which [articles you've read](#), [speeding up Bitcoin clients](#), [detecting malicious web sites](#), and [improving the performance of caches](#).

This page will help you choose an optimal size for your filter, or explore how the different parameters interact.

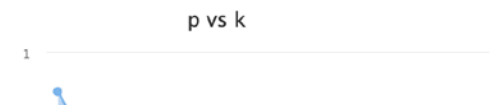
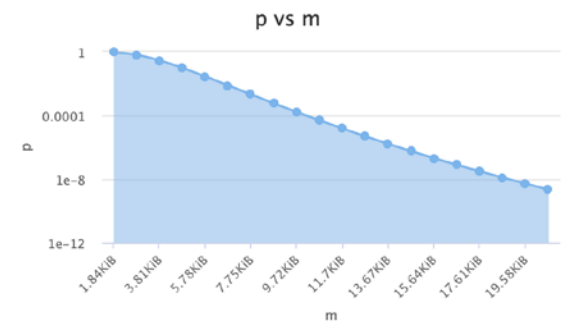
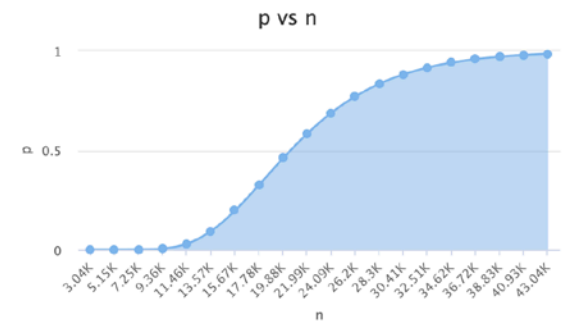
$n$  Number of items in the filter (optionally with SI units: k, M, G, T, P, E, Z, Y)

$p$  Probability of false positives, fraction between 0 and 1 or a number indicating 1-in- $p$

$m$  Number of bits in the filter (or a size with KB, KiB, MB, Mb, GiB, etc)

$k$  Number of hash functions

$n = 4,000$   
 $p = 0.0000001$  (1 in 9,994,297)  
 $m = 134,191$  (16.38KiB)  
 $k = 23$



---

## Trade off memory space vs. speed

- Larger bloom filter:
  - Reduces false positives → fewer reads to disk → lower latency, higher throughput
- But...
  - Takes up more space in memory → less space for caching database entries in memory → higher chance of going to disk → higher latency, lower throughput

## Example of space vs. speed trade off

- Example: database has 2GB of memory for caching and 100GB of flash
  - Needs to store 1 billion entries (each ~100B)

- Flash access is 100us, memory access is 100ns

- Which bloom filter parameters would you choose?

1. 600MB bloom filter:

- Bloom filter false positive rate of 9%
- 1.4GB of DRAM left → cache hit rate of 90%

2. 1.1GB bloom filter:

- Bloom filter false positive rate of 1%
- 0.9GB of DRAM left → cache hit rate of 70%

- 60% of requests return object does not exist:

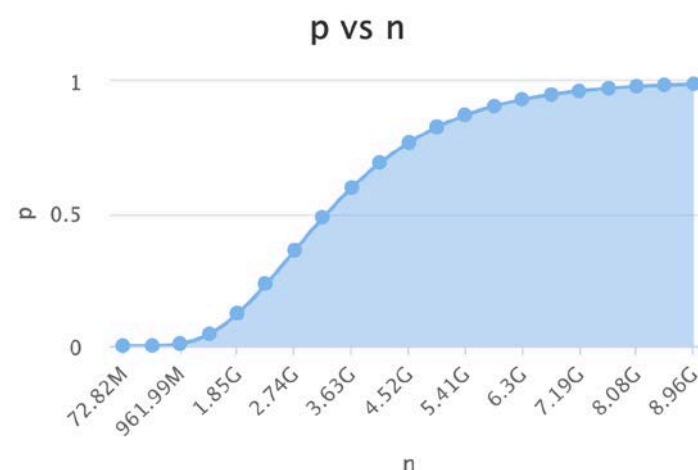
- $avg\ latency = \mathbb{P}(exists) \cdot (\mathbb{P}(cached) \cdot 100ns + \mathbb{P}(not\_cached) \cdot 100\mu s) + \mathbb{P}(doesn't\_exist) \cdot (\mathbb{P}(true\ positive) \cdot 100ns + \mathbb{P}(false\ positive) \cdot 100\mu s)$

- Scenario 1

- $0.4 \cdot (0.9 \cdot 100ns + 0.1 \cdot 100\mu s) + 0.6 \cdot (0.91 \cdot 100ns + 0.09 \cdot 100\mu s) = \mathbf{9.49\mu s}$

- Scenario 2:

- **12.64us**



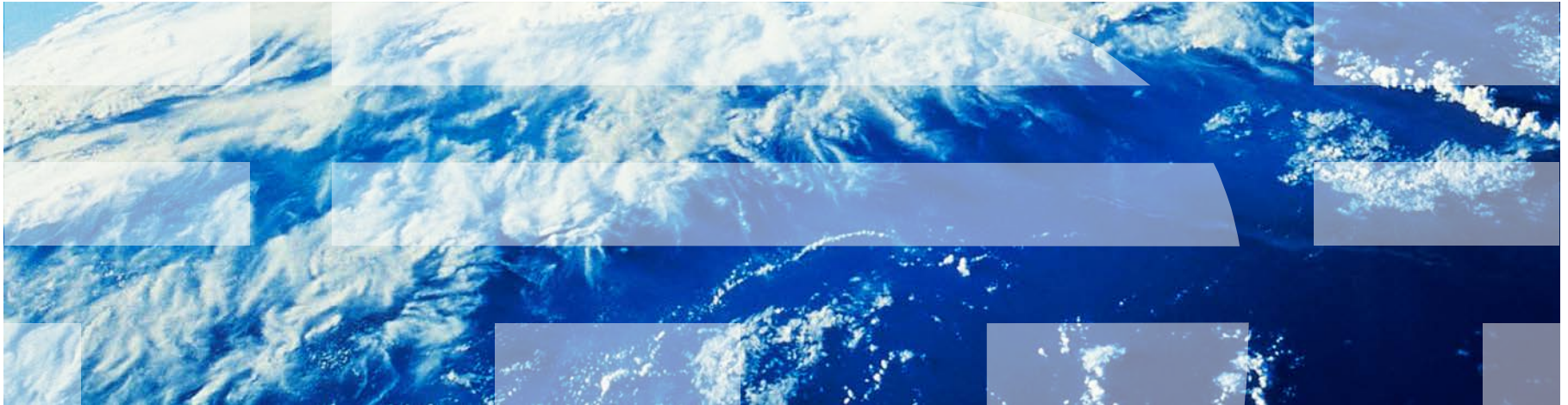
---

## Other issues with bloom filters

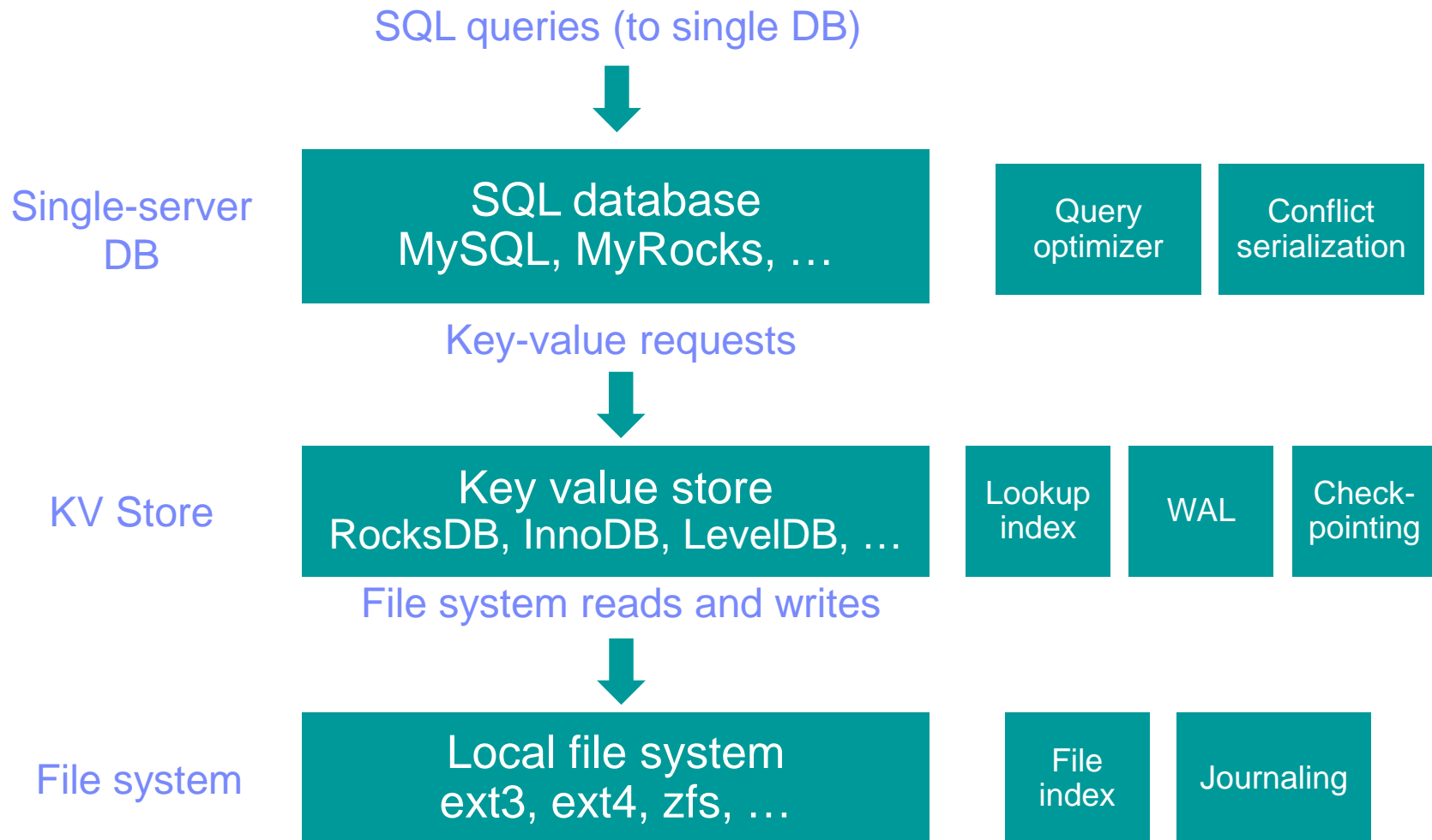
- Can get “depleted” over time
- Need to estimate number of unique entries in advance
- Do not support deletes!
  - Why?
- Improvements: counting bloom filters, cuckoo filters, learned bloom filters, elastic bloom filters... This is a hot research area!

---

# Database Architecture

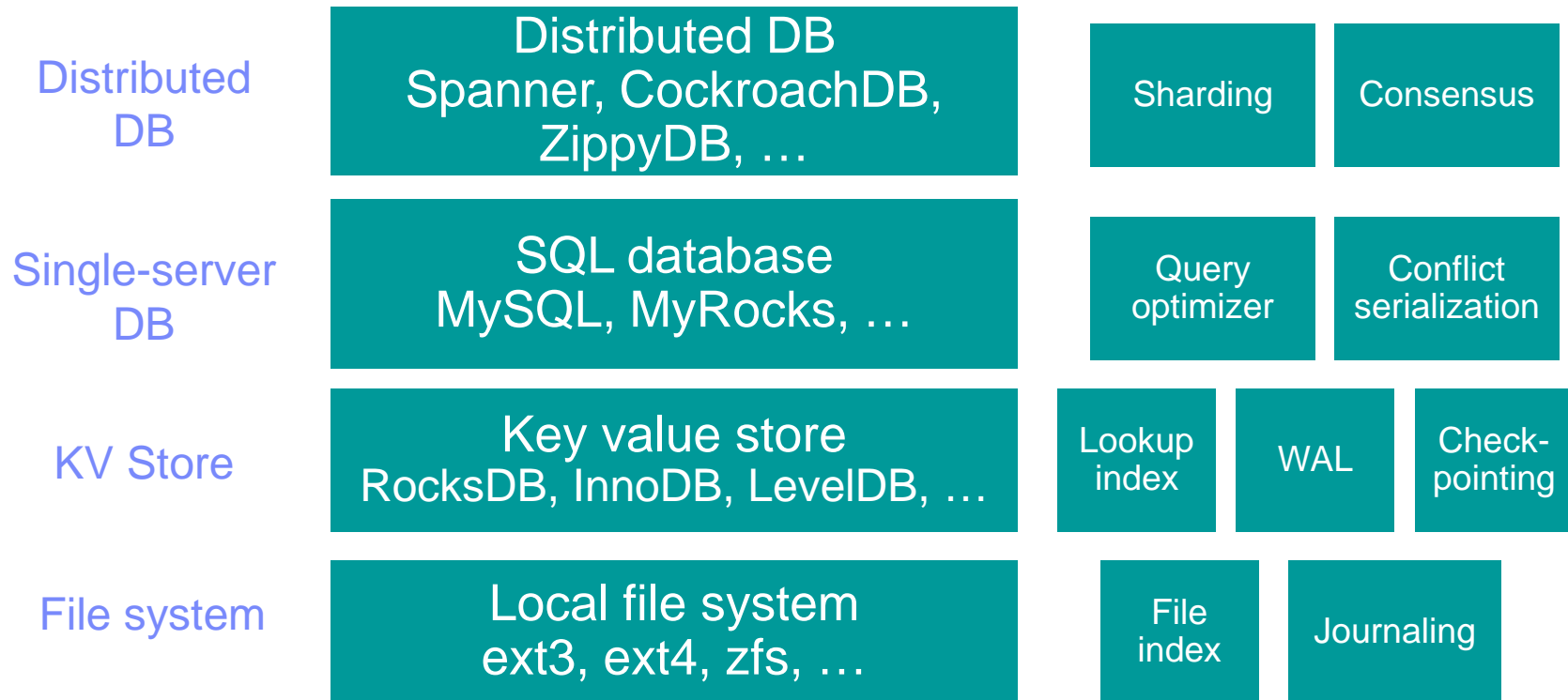


## Modern databases often consist of several layers



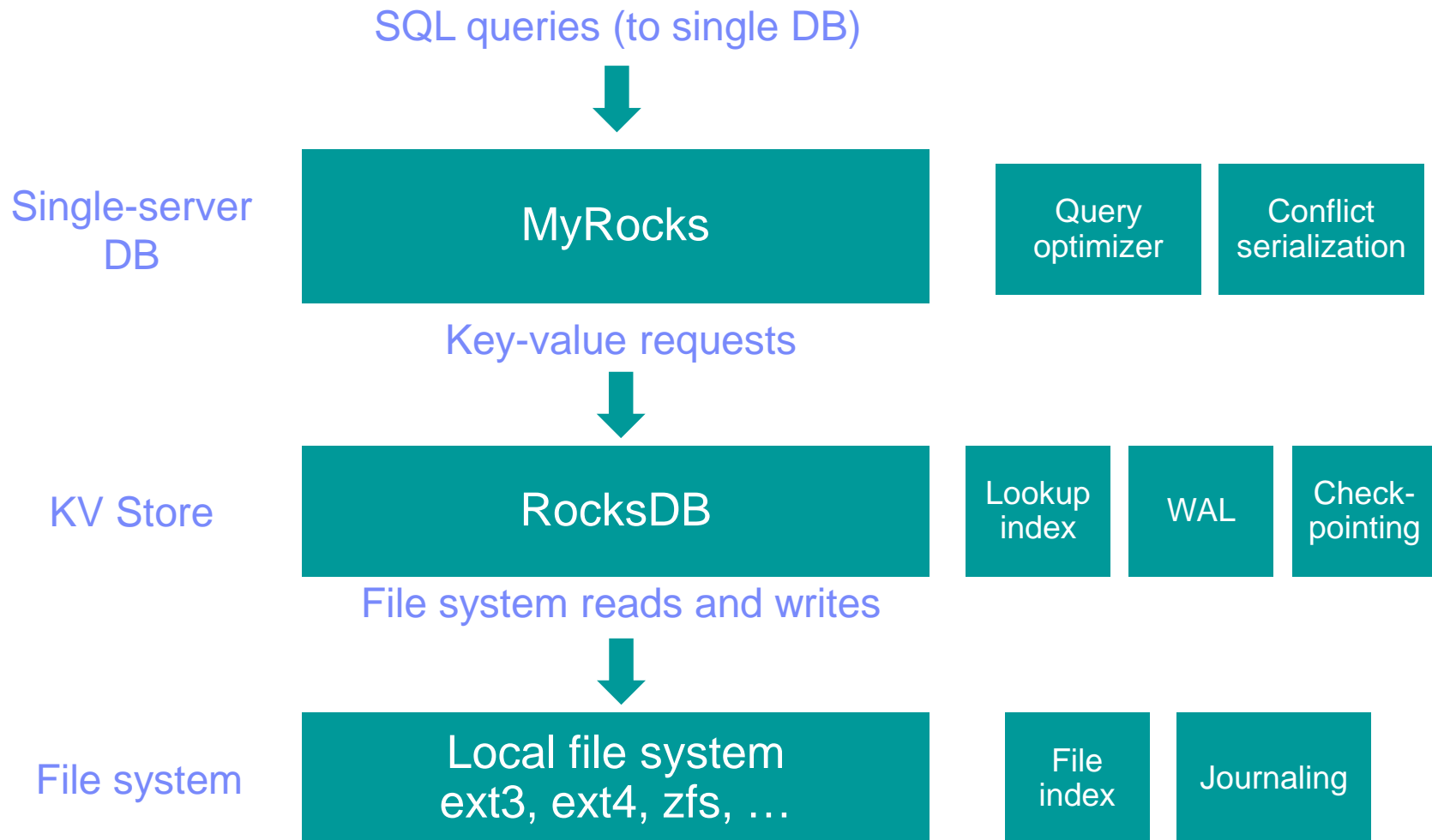
## Distributed databases add even another layer

We'll talk more about distributed computer systems later...





## We'll focus on one example database: MyRocks + RocksDB



---

## Very brief overview of local file systems

- This is the file system running on the server itself
  - Similar to the file system on your computer!
- Maps physical storage into files, which can be read/written by applications
- Caches some data in memory (file system cached is called *page cache*)
- Logging (similar to WAL) called *file system journal*
- Databases (and other big data systems) often disable many features of file system
  - Don't care about directories
  - Don't rely on file system journal for maintaining ACID
  - Often want to manage their own cache

---

## Key-value Stores

- Databases with a really simple interface
  - Do not support SQL
- Get (key)
- Put (key, value)
- Delete (key)
- Sometimes:
  - Update (key, value)
  - Multiget (key1, key2, key3, ...)
  - Get\_range ([key\_i, ... key\_j])
- Key-value (KV) stores can be used on their own (NoSQL) or can be a foundation for an ACID SQL database

## Showcase: RocksDB

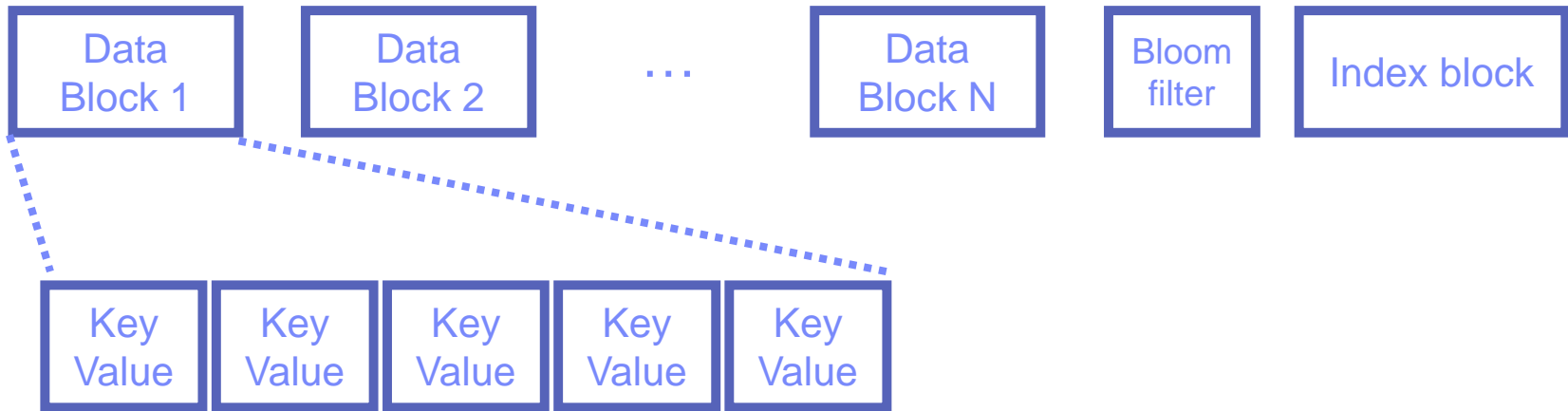
- Open source KV store built by Facebook
  - Based on another open source KV store, LevelDB, built by Google
- Optimized for flash
  - Large contiguous writes
  - Generates a relatively small number of writes
- Optimized for range queries
  - Applications often need contiguous range of keys (e.g., get an entire column from a database)

---

# RocksDB components

- Memtable
  - In-memory data structure
  - Stores all incoming writes
    - Why is this a good idea?
  - All writes go to Memtable
  - Small: 64/128MB
- Logfile
  - Write ahead log
  - Sequentially written to
  - Stored persistent in storage (not memory)
- SSTfile
  - Data structure that stores the contents of the database in storage
  - A file that contains a set of **sorted** key-value pairs (keys + values)
  - Organized in **levels**
  - Immutable (i.e., write-only, can never be updated)
    - Why is this a good idea?
  - Sorted data → makes it easier to lookup keys

## SSTfile structure



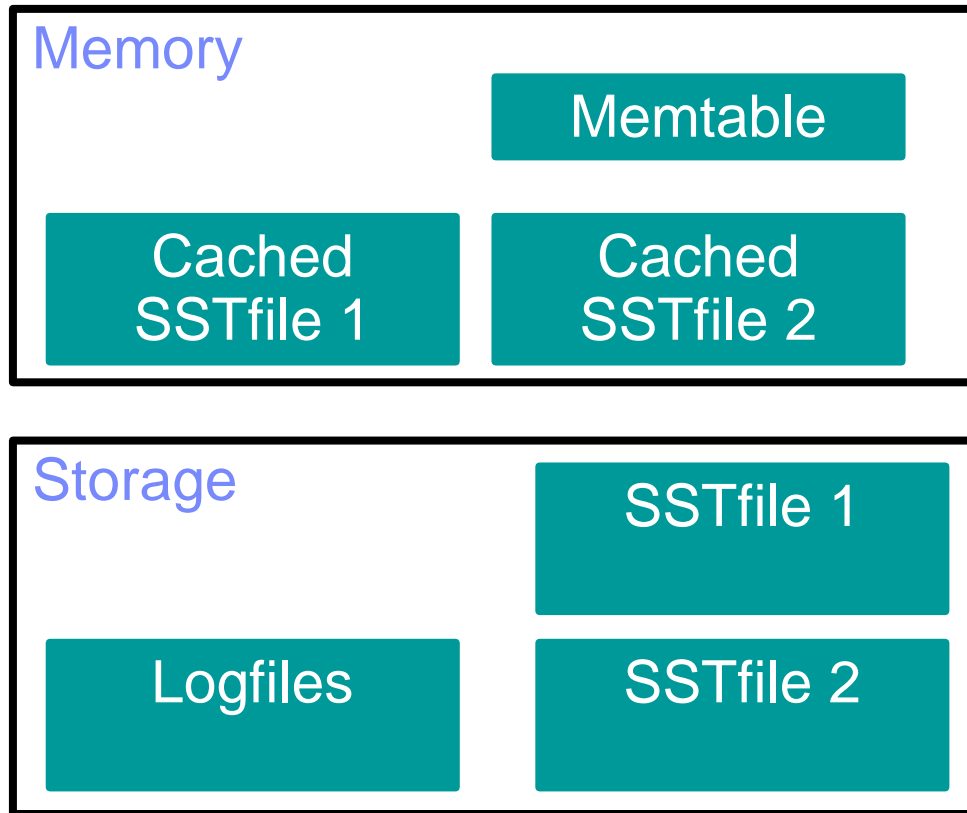
---

## SSTfiles and Memtables

- On-disk SSTfiles are always loaded into memory before they are read
- All **writes** go directly to Memtable
- **Reads** check the Memtable first, then the SSTfiles indices
- SSTfile indices and bloom filter are usually cached permanently in DRAM (if they fit)
- Periodically Memtable is **flushed** to disk
- Periodically old on-disk SSTfiles are **merged/compacted** to create new SSTfiles
  - Recall, SSTfiles are immutable, so we need a way to remove stale data (updated/deleted values)
  - Stored in a data structure called **Log-structure Merge Tree (LSM-Tree)**



## Simplified architecture



### SSTfile index

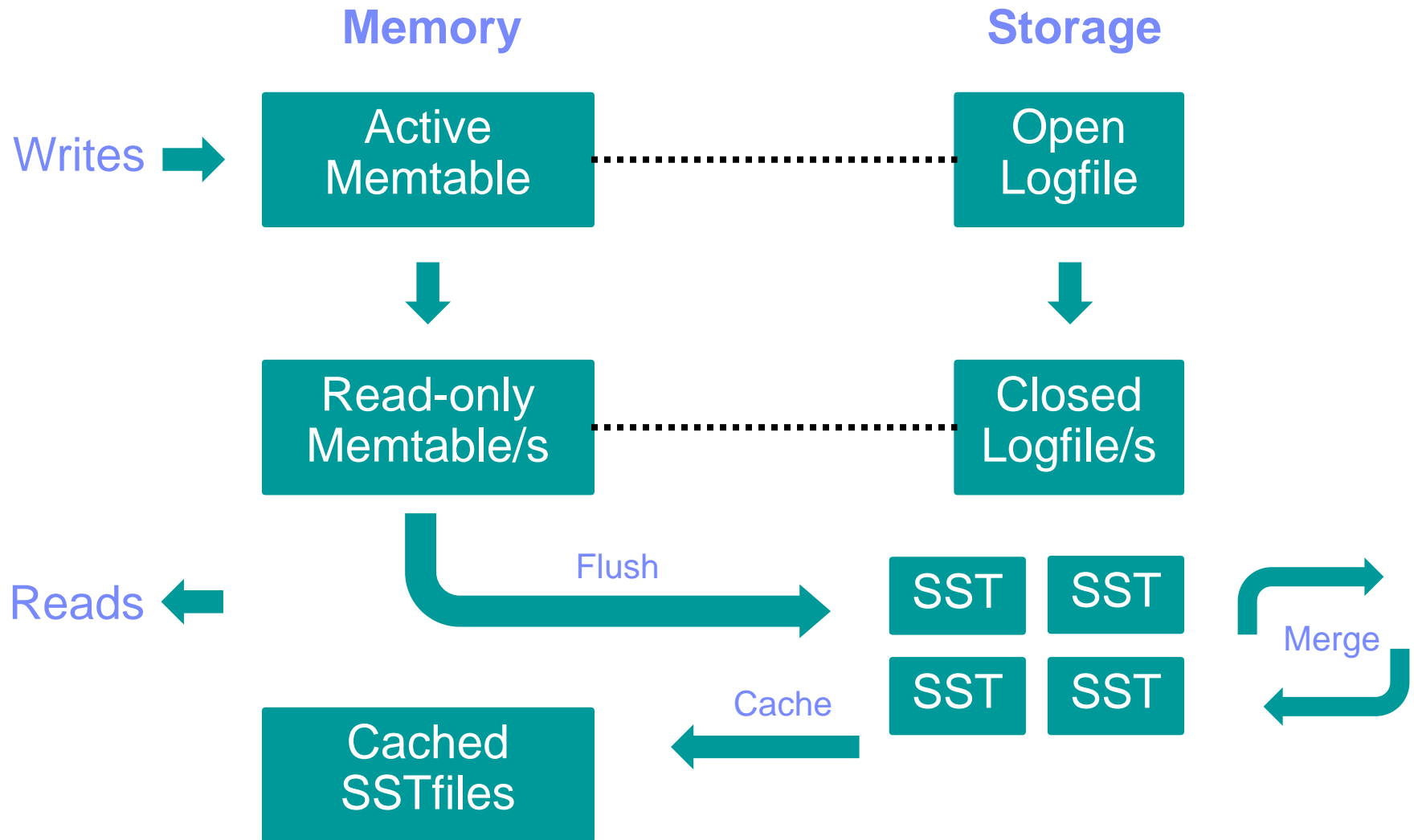
Key	Offset
Key	Offset
...	...

### Simplified SSTfile data

Key	Value	Key	Value	...
-----	-------	-----	-------	-----



## Basic functions

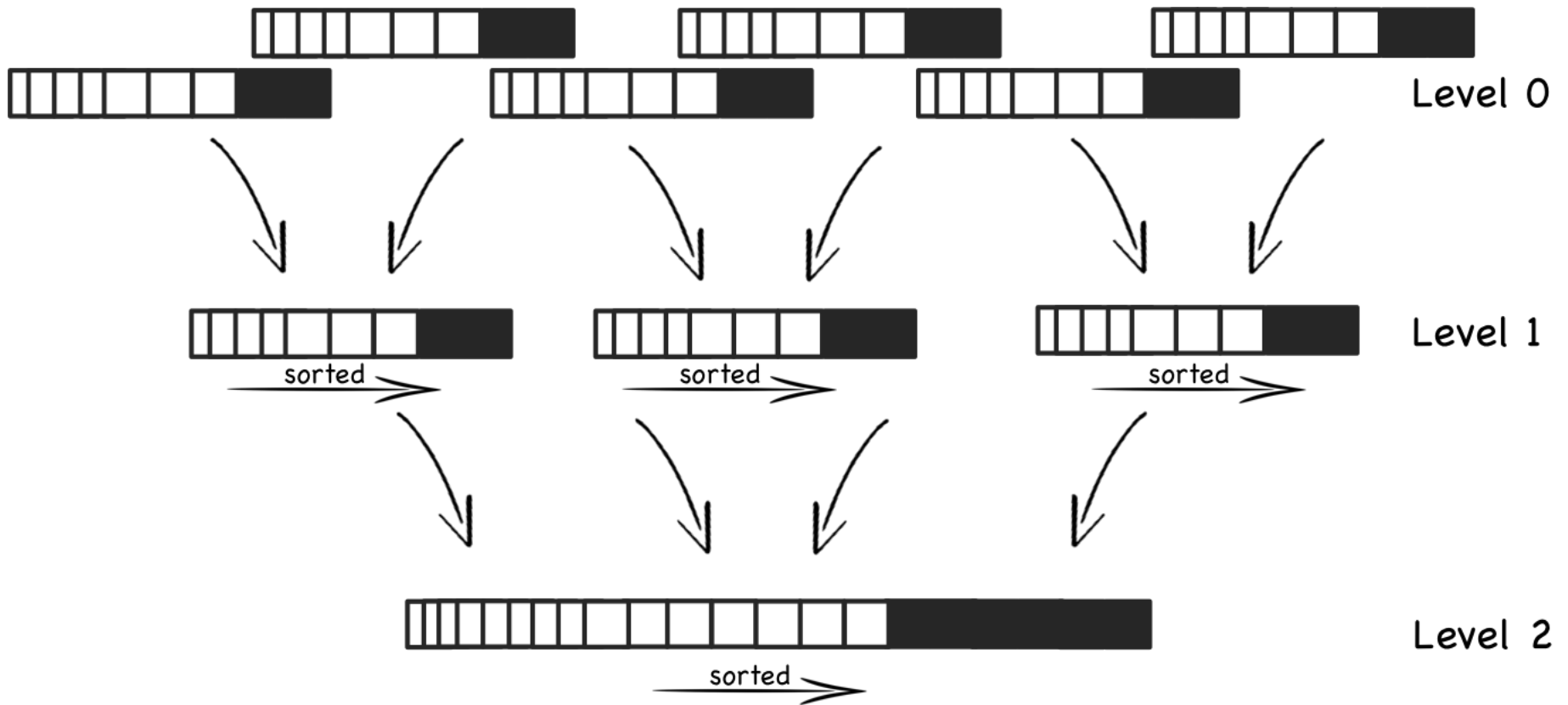


---

## How does RocksDB minimize random writes

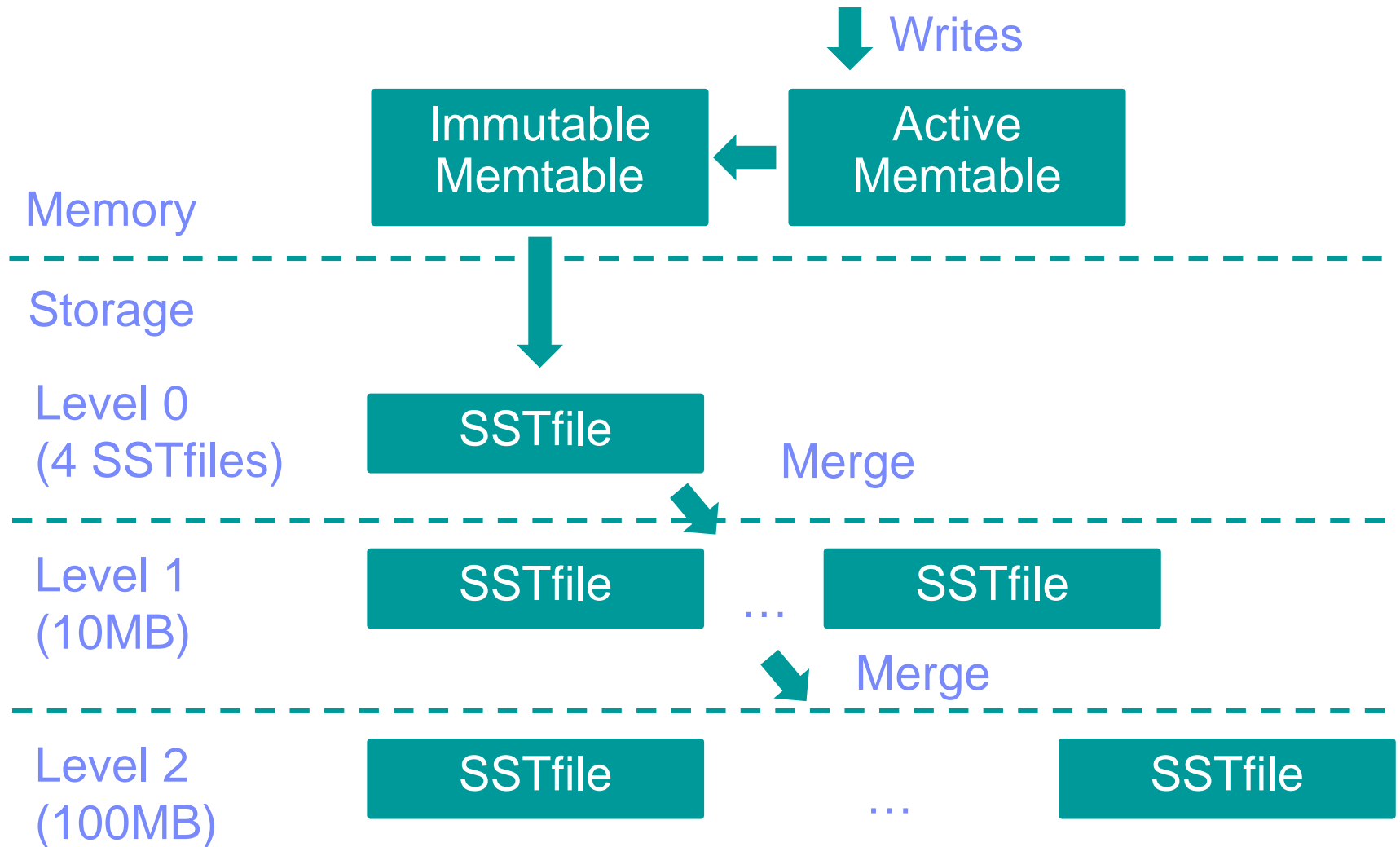
- Data write (insert/update)
  - New data written to memory (Memtable) and to logfile sequentially
  - Memtable fills up → flushed to SSTfile on disk
- Data read
  - Memtable in memory
  - SSTfile indices in memory
    - Use bloom filters to determine existence

## Log-structured Merge Tree (LSM-Tree)



Compaction continues creating fewer, larger and larger files

# RocksDB LSM-Tree Architecture



---

## LSM-Trees

- Most recently written data is at the top of the tree
  - Newer versions are in a higher level than older version
    - Why? Because older versions get compacted
  - LSM-Tree is searched in order from the highest level
- In RocksDB, highest level is in memory
  - Not sorted
  - Memory has good random write performance (not the case for flash/magnetic disk)
- Data that ends up in the lower levels is not updated frequently

---

## Merging

- Background process periodically merges SSTfiles
  - Parallel computations on different parts of the DB occur simultaneously (using locks)
- Merges SSTfiles from higher level to larger SSTfile in lower level
  - Removes old versions of the same key
  - Removes deleted or overwritten keys
- Each level is 10 times larger than the previous
- In level 0 different SSTfile might contain overlapping key ranges
- In levels 1-n SSTfile key range are not overlapping
  - For example:
    - SSTfile 1: key range: [0, 5]
    - SSTfile 2: key range [6, 202]
    - SSTfile 3: key range [205, 421]
- Every level is 10 times larger than the other
  - Most data sits at the bottom level

---

## Concurrency

- Database maintains a lock table
- Every update acquires a lock beforehand
- Actively check for conflicts

---

## MyRocks (SQL and ACID over RocksDB)

- Implements ACID on top of RocksDB
  - Uses RocksDB locks to implement isolation
- Implements SQL over RocksDB API
  - Translates rows/columns to gets/puts/multiget